第6章

从客户端到云的并行处理器

付俊宁 吕卫 译

引言

- 目标：连接多台计算机以获得更高性能
  - 多处理器
  - 伸缩性、可用性、功耗效率
- 任务级（进程级）并行
  - 处理独立任务时具有高吞吐率
- 并行处理程序
  - 在多个处理器上运行的单一程序
- 多核微处理器
  - 有多个处理器（核）的芯片

硬件和软件

- 硬件
  - 串行：例如，Pentium 4
  - 并行：例如，四核Xeon e5345
- 软件
  - 顺序：例如，矩阵乘法
  - 并发：例如，操作系统
- 顺序/并发软件可以在串行/并行硬件上运行
  - 挑战：让并行硬件得到高效利用

回顾已学内容

- §2.11：并行与指令
  - 同步
- §3.6：并行性和计算机算术
  - 子字并行
- §4.10：并行与高级指令级并行
- §5.10：并行与存储器层次结构
  - cache一致性

并行编程

- 并行软件是困难所在
- 需要获得显著的性能提升
  - 否则，用一个更快的单处理器就行了，因为更省事儿！
- 困难
  - 划分
  - 协调
  - 通信开销

Amdahl定律

- 顺序部分会限制加速比
- 例：100个处理器，90倍的加速比？
  - $T_{改进后} = T_{可并行}/100 + T_{顺序}$
  - 加速比 $= \dfrac{1}{(1 - F_{可并行}) + F_{可并行}/100} = 90$
  - 解得：$F_{可并行} = 0.999$
- 需要顺序部分仅占本来用时的0.1%

比例缩放的例子

- 负载：10个标量求和，及10×10矩阵求和
  - 10~100个处理器的加速比
- 单个处理器：时间 = (10 + 100)×$t_{加法}$
- 10个处理器
  - 时间 = 10×$t_{加法}$ + 100/10×$t_{加法}$ = 20×$t_{加法}$
  - 加速比 = 110/20 = 5.5（潜在加速比的55%）
- 100个处理器
  - 时间 = 10×$t_{加法}$ + 100/100×$t_{加法}$ = 11×$t_{加法}$
  - 加速比 = 110/11 = 10（潜在加速比的10%）
- 假定处理器间负载均衡

比例缩放的例子（续）

- 如果矩阵维数是100×100呢？
- 单个处理器：时间 = (10 + 10000)×$t_{加法}$
- 10个处理器
  - 时间 = 10×$t_{加法}$ + 10000/10×$t_{加法}$ = 1010×$t_{加法}$
  - 加速比 = 10010/1010 = 9.9（潜在加速比的99%）
- 100个处理器
  - 时间 = 10×$t_{加法}$ + 10000/100×$t_{加法}$ = 110×$t_{加法}$
  - 加速比 = 10010/110 = 91（潜在加速比的91%）
- 假定负载均衡

强比例缩放与弱比例缩放

- 强比例缩放：问题规模固定
  - 比如前面的例子
- 弱比例缩放：问题规模与处理器个数成正比
  - 10个处理器，10×10矩阵
    - 时间 = $20 \times t_{加法}$
  - 100个处理器，32×32矩阵
    - 时间 = $10 \times t_{加法} + 1000/100 \times t_{加法} = 20 \times t_{加法}$
  - 在本例中的性能恒定

指令与数据流

- 另一种分类方式

		数据流	
		单	多
指令流	单	**SISD：** Intel Pentium 4	**SIMD：** x86的SSE指令
	多	**MISD：** 至今没有实例	**MIMD：** Intel Xeon e5345

- SPMD：单程序多数据
  - MIMD计算机上的一个并行程序
  - 用条件代码控制不同处理器

向量处理器

- 高度流水的功能单元
- 数据在向量寄存器和功能单元之间流动
  - 数据从存储器收集到寄存器
  - 结果从寄存器保存到存储器
- 例：RISC-V的向量扩展
  - v0~v31：32个64数据元的寄存器，数据元宽度为64位
  - 向量指令
    - fld.v, fsd.v：装载/保存向量
    - fadd.d.v：将双精度向量相加
    - fadd.d.vs：将标量加到双精度向量的每个数据元上
- 显著降低取指所用带宽

例：DAXPY (Y = a×X + Y)

- 传统RISC-V代码：

```
        fld     f0,a(x3)      // load scalar a
        addi    x5,x19,512    // end of array X
loop:   fld     f1,0(x19)     // load x[i]
        fmul.d  f1,f1,f0      // a * x[i]
        fld     f2,0(x20)     // load y[i]
        fadd.d  f2,f2,f1      // a * x[i] + y[i]
        fsd     f2,0(x20)     // store y[i]
        addi    x19,x19,8     // increment index to x
        addi    x20,x20,8     // increment index to y
        bltu    x19,x5,loop   // repeat if not done
```

- 向量RISC-V代码：

```
        fld       f0,a(x3)    // load scalar a
        fld.v     v0,0(x19)   // load vector x
        fmul.d.vs v0,v0,f0    // vector-scalar multiply
        fld.v     v1,0(x20)   // load vector y
        fadd.d.v  v1,v1,v0    // vector-vector add
        fsd.v     v1,0(x20)   // store vector y
```

向量 vs. 标量

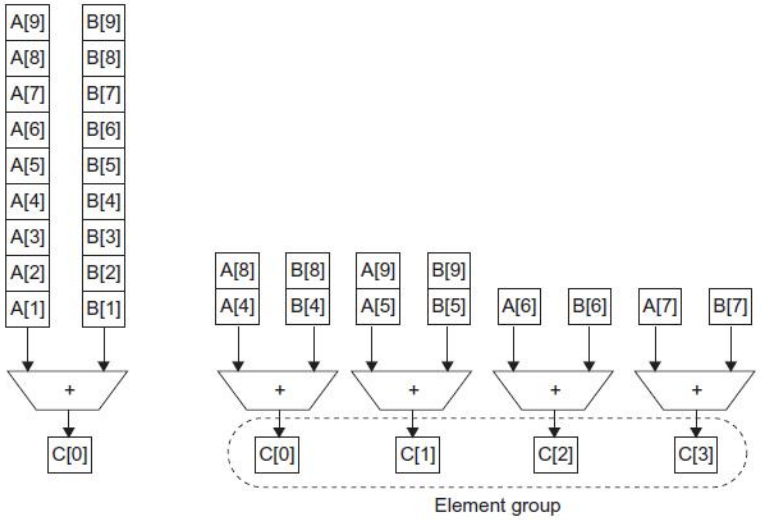
- 向量体系结构与编译器
  - 简化了数据并行编程
  - 显式的声明消除了循环导致的相关性
    - 减少硬件要进行的检查
  - 规则的存取模式可受益于交叉式突发存储器
  - 避免使用循环，从而避免了控制冒险
- 比凭空增加的多媒体扩展更为通用（如MMX、SSE）
  - 更好地适应编译器技术

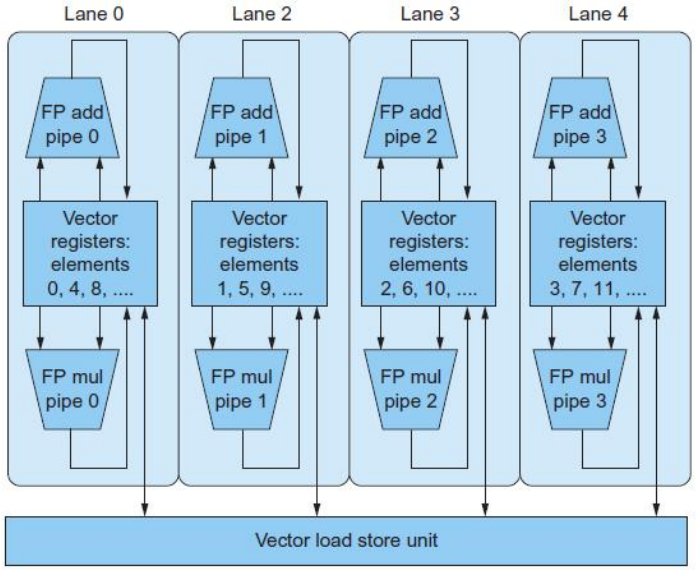
单指令多数据(SIMD)

- 逐元素地操作数据向量
  - 例如，x86中的MMX和SSE指令
    - 128位宽的寄存器中有多个数据元素
- 所有处理器在同一时间执行相同指令
  - 各自对应不同的数据地址等等
- 简化了同步
- 减少了指令控制硬件
- 最适合高度数据并行的应用

向量 vs. 多媒体扩展

- 向量指令的向量宽度可变，多媒体扩展宽度固定
- 向量指令支持按步长存取，多媒体扩展不支持
- 向量单元可以是流水线式和阵列式功能单元的组合





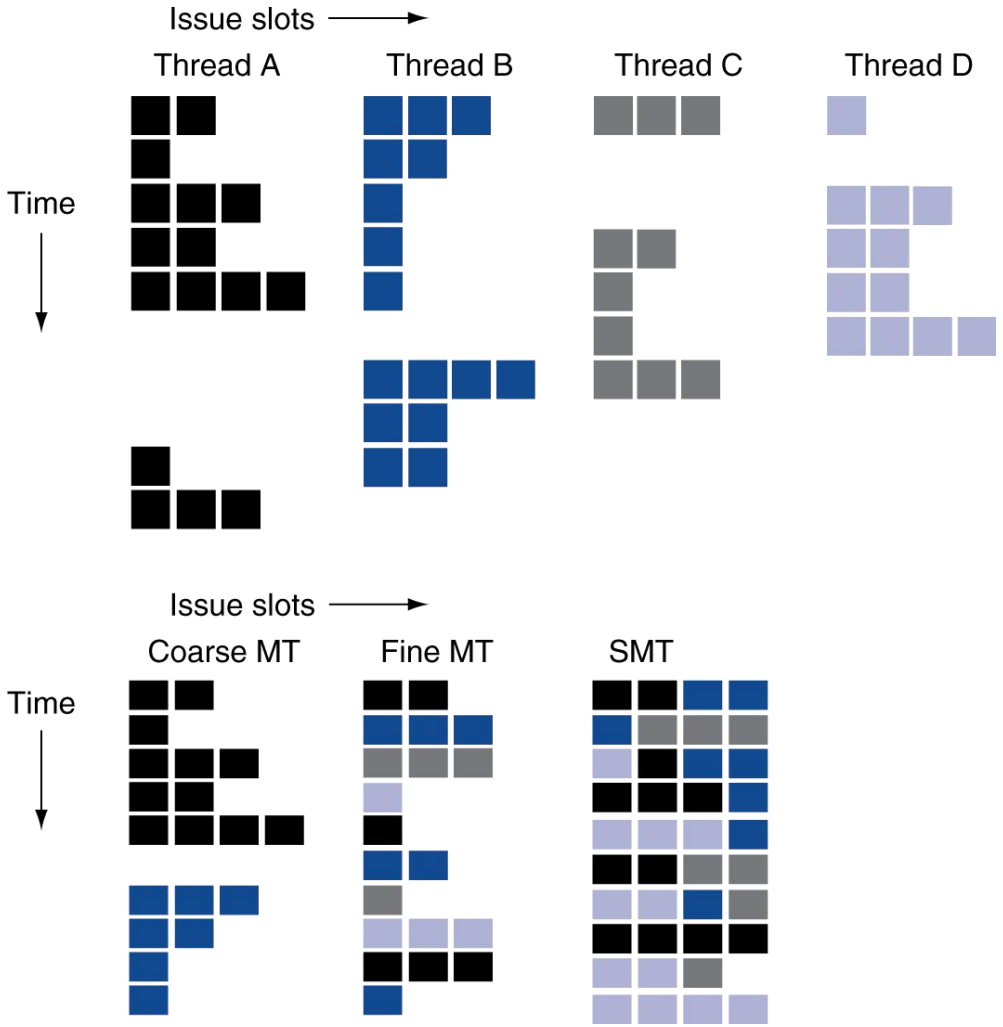
多线程(Multithreading)

- 并行执行多个线程
  - 复制寄存器、PC等等
  - 在线程间快速切换
- 细粒度多线程
  - 每个时钟周期之后切换线程
  - 交叉执行指令
  - 如果某一线程阻塞，执行其他线程
- 粗粒度多线程
  - 仅在长时间阻塞时切换（例如，L2-cache缺失）
  - 简化了硬件，但无法隐藏短阻塞（例如，数据冒险）

同时多线程

- 在多发射动态调度的处理器中
  - 调度多个线程中的指令
  - 在功能单元就绪时执行来自不相关线程的指令
  - 在线程内部，通过调度和寄存器重命名处理相关性
- 例：Intel Pentium-4 HT
  - 双线程：复制寄存器，共享功能单元和cache

多线程的例子

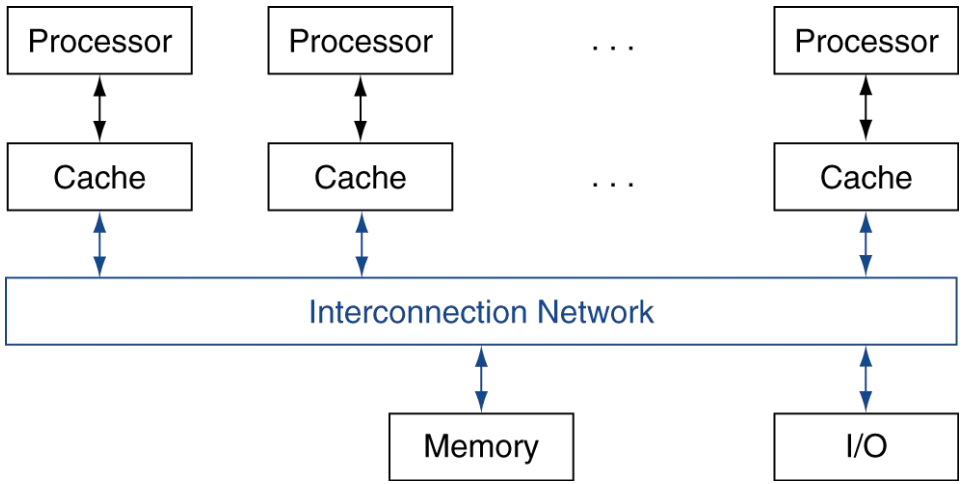

Issue slots
Thread A
Thread B
Thread C
Thread D
Time
Issue slots
Coarse MT
Fine MT
SMT
Time

多线程的未来

- 它会存在下去吗？以什么样的形式？
- 功耗考虑 ⇒ 简化的微架构
  - 更简单的多线程形式
- 容许cache缺失延迟
  - 线程切换也许是最有效的
- 多个简单核也许能更有效地共享资源

共享内存

- SMP：共享内存多处理器
  - 硬件为所有处理器提供单一物理地址空间
  - 用锁来同步共享变量
  - 存储器访问时间
    - UMA（统一）与NUMA（非统一）



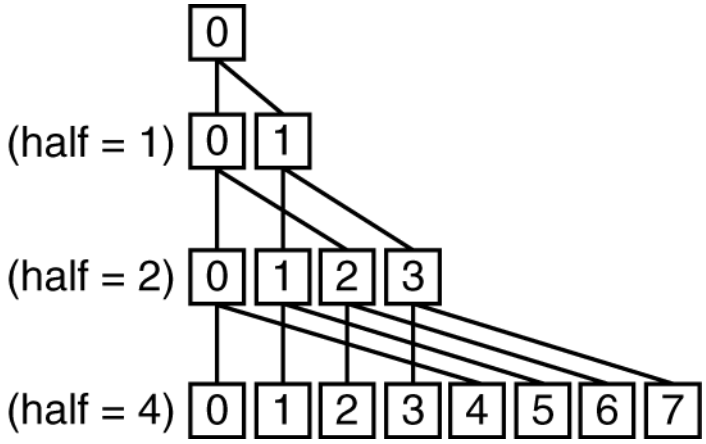
例：约简求和

- 用64个UMA的处理器给64,000个数求和
  - 每个处理器都有ID：0 ≤ Pn ≤ 63
  - 分给每个处理器1000个数
  - 先用每个处理器求和

```
sum[Pn] = 0;
for (i = 1000*Pn;
     i < 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- 现在需要将部分和相加
  - 约简：分而治之
  - 用一半的处理器给成对的数求和，然后用四分之一，…
  - 需要在约简步骤之间同步

例：约简求和

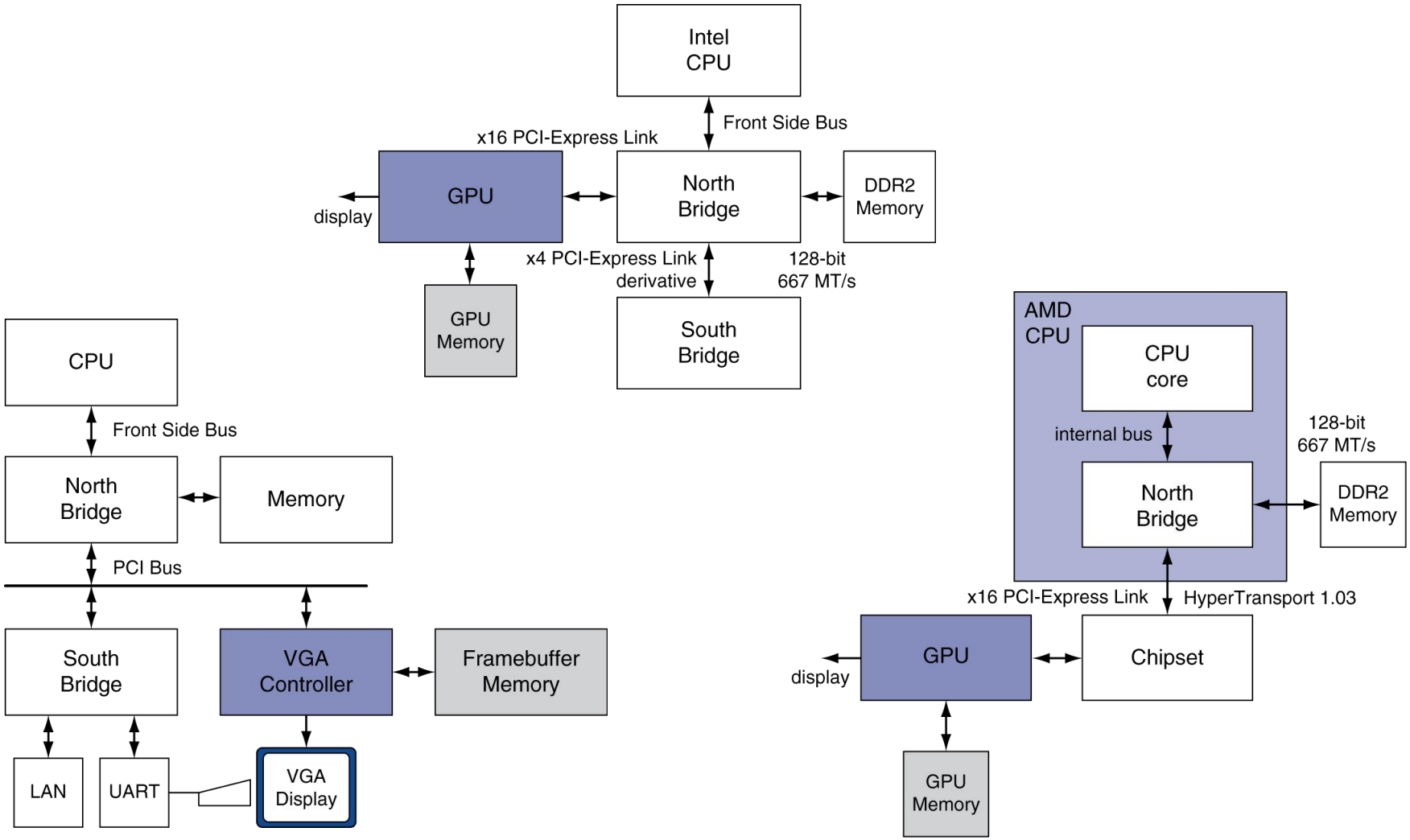



```
half = 64;
do
  synch();
  if(half%2 != 0 && Pn == 0)
    sum[0] += sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if(Pn < half) sum[Pn] += sum[Pn+half];
while(half > 1);
```

GPU的历史

- 早期显卡
  - 具备用于视频输出的地址发生功能的帧缓存
- 3D图形处理
  - 起初用于高端计算机（例如，SGI）
  - 摩尔定律 ⇒ 更低造价、更高密度
  - PC和游戏终端用的3D显卡
- 图形处理单元(Graphics Processing Unit)
  - 针对3D图形任务的处理器
  - 顶点/像素处理、着色、纹理映射、光栅化

系统中的显卡

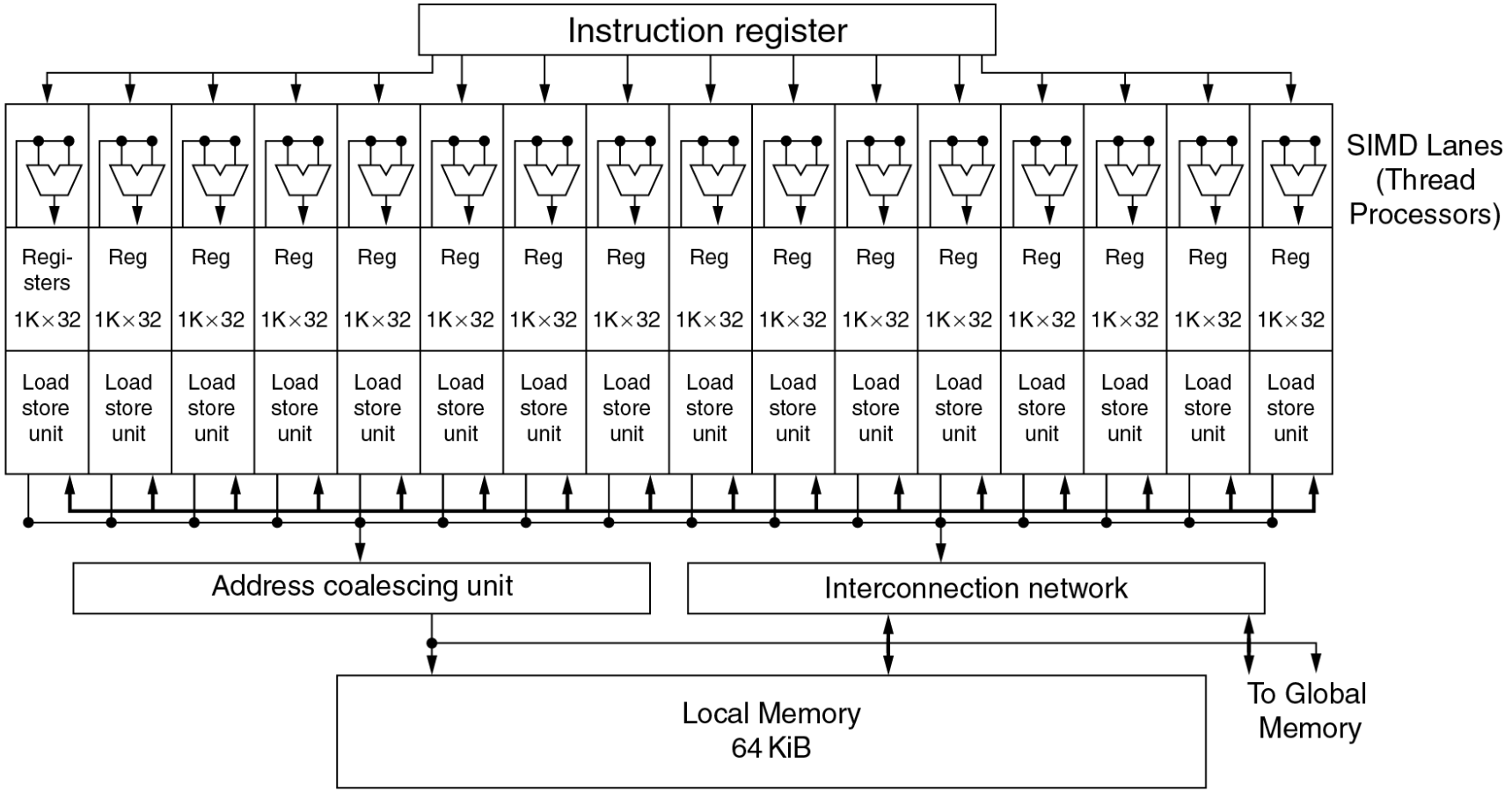

Intel
CPU
Front Side Bus
x16 PCI-Express Link
GPU
North
Bridge
DDR2
Memory
display
x4 PCI-Express Link
derivative
128-bit
667 MT/s
GPU
Memory
South
Bridge
CPU
Front Side Bus
North
Bridge
Memory
PCI Bus
South
Bridge
VGA
Controller
Framebuffer
Memory
LAN
UART
VGA
Display
AMD
CPU
CPU
core
internal bus
128-bit
667 MT/s
North
Bridge
DDR2
Memory
x16 PCI-Express Link
HyperTransport 1.03
display
GPU
Chipset
GPU
Memory

GPU体系结构

- 高度数据并行的处理
  - GPU是高度多线程的
  - 通过线程切换隐藏存储器延迟
    - 对多级cache的依赖更小
  - 图形存储器（显存）的位宽和带宽更大
- 向通用GPU发展
  - 异构CPU/GPU系统
  - CPU处理顺序代码，GPU处理并行代码
- 编程语言/API
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

例：NVIDIA Fermi

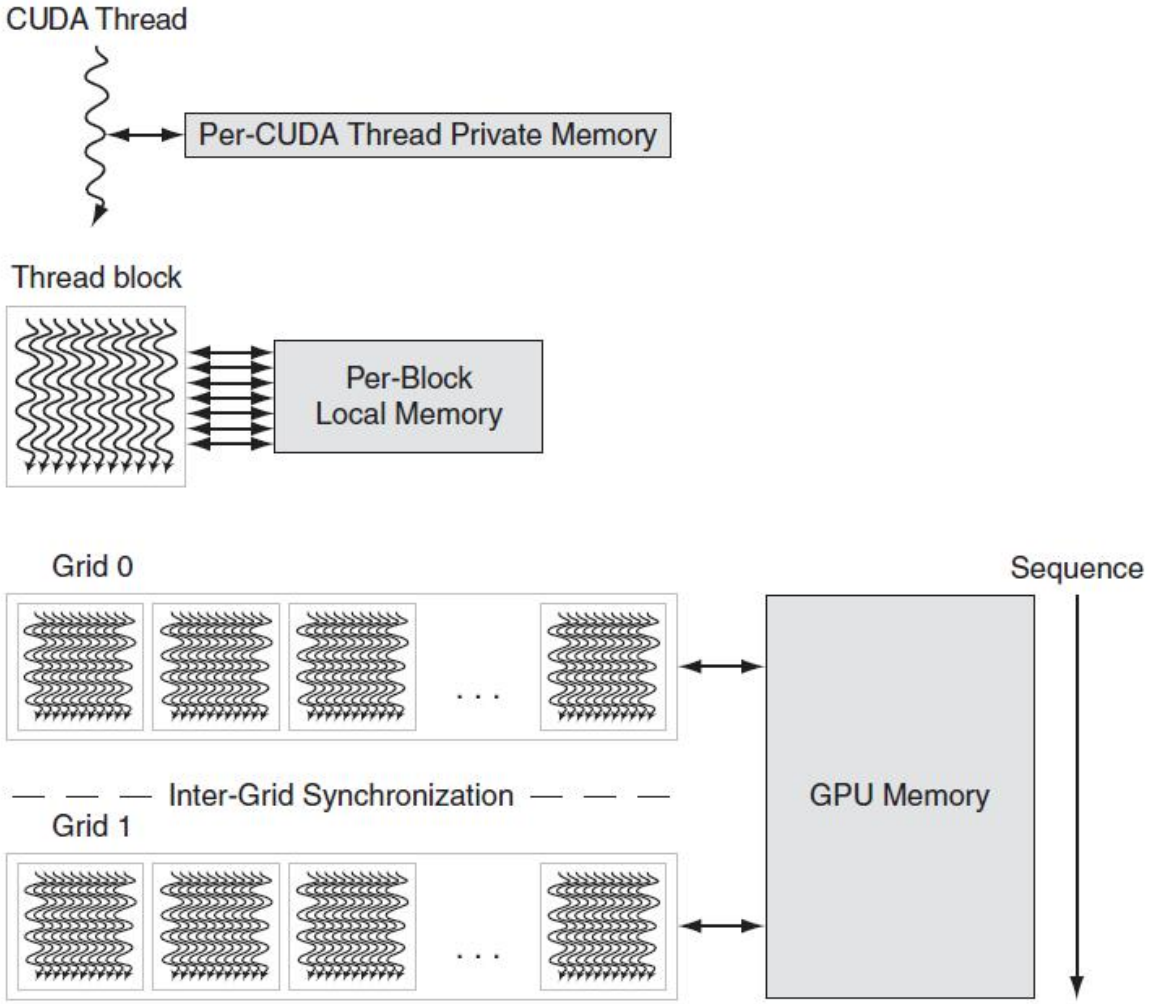
- 多个SIMD处理器，每个如图所示



例：NVIDIA Fermi

- SIMD处理器：16条SIMD通道
- SIMD指令
  - 在32个线程上进行运算，每个线程处理1个数据元
  - 在16条通道的处理器上动态调度，用2个时钟周期
- 32K个32位宽度的寄存器分布在各通道上
  - 每个线程的上下文有64个寄存器

GPU存储器结构


CUDA Thread
Per-CUDA Thread Private Memory
Thread block
Per-Block
Local Memory
Grid 0
Sequence
Inter-Grid Synchronization
Grid 1
GPU Memory

给GPU归类

- 不太适合用SIMD/MIMD模型
  - 在线程中条件执行，如同MIMD
    - 但性能有下降
    - 编写通用代码时需小心

	静态：编译时发现	动态：运行时发现
指令级并行	VLIW	超标量
数据级并行	SIMD或向量机	**Tesla多处理器**

GPU透视

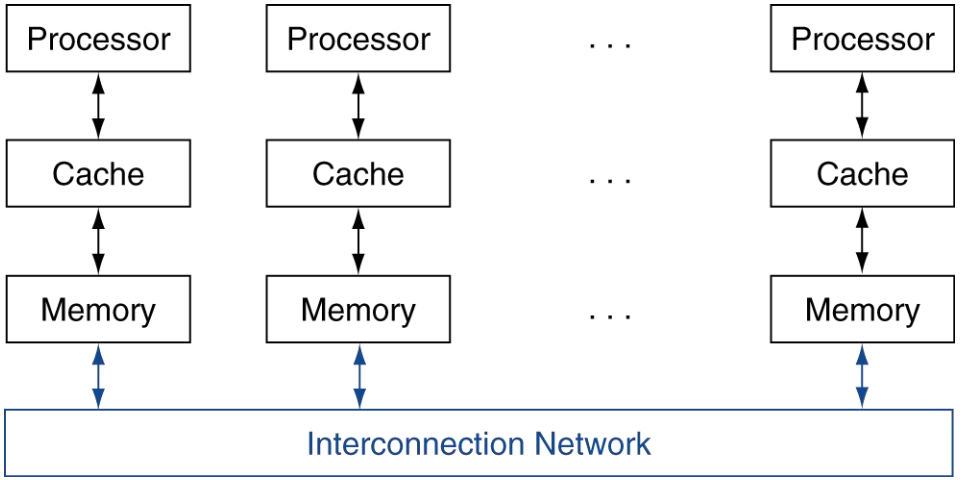
特性	SIMD多核	GPU
SIMD处理器数目	4 ~ 8	8 ~ 16
SIMD通道数目/处理器	2 ~ 4	8 ~ 16
支持SIMD线程的硬件数量	2 ~ 4	16 ~ 32
单精度对双精度的典型性能比	2:1	2:1
最大的cache容量	8 MB	0.75 MB
存储器地址大小	64位	64位
存储器大小	8 GB ~ 256 GB	4 GB ~ 6 GB
页级存储器保护	有	无
需要分页	是	否
cache一致性	有	无

GPU术语指南

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

消息传递

- 每个处理器有其私有物理地址空间
- 硬件在处理器之间收发消息



松散耦合的集群

- 独立计算机构成的网络
  - 每台计算机有私有的存储器和OS
  - 使用I/O系统相联
    - 例如，以太网/交换机、互联网
- 适用于处理独立任务的应用
  - Web服务器、数据库、仿真…
- 高可用性、伸缩性，价格可接受
- 问题
  - 管理成本（不如虚拟机）
  - 互联带宽低
    - 对比SMP的处理器/存储器带宽

再论约简求和

- 用64个处理器对64,000个数求和
- 先给每个处理器分配1000个数
  - 求部分和的循环

```
sum = 0;
for(i = 0; i<1000; i = i + 1)
  sum = sum + AN[i];
```

- 约简
  - 一半处理器发送，另一半接收并求和
  - 四分之一处理器发送，四分之一接收并求和...

再论约简求和

- 给定send()和receive()操作

```
limit = 64; half = 64;/* 64 processors */
do
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if(Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if(Pn < (limit/2))
    sum += receive();
  limit = half; /* upper limit of senders */
while(half > 1); /* exit with final sum */
```

  - send/receive亦提供同步
  - 假定send/receive和加法运算时间近似

网格计算(Grid Computing)

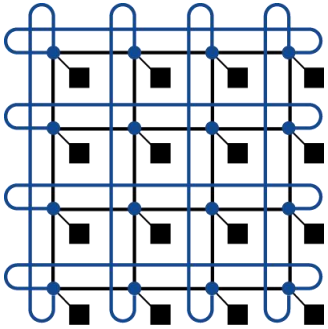
- 通过长距离网络互联的独立的计算机
  - 例如，互联网连接
  - 工作单元分包，结果传回
- 可以将PC的空闲时间利用起来
  - 例如，SETI@home、World Community Grid

互联网络

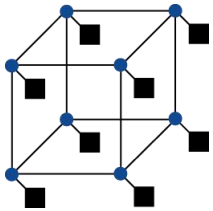
- 网络拓扑
  - 处理器、开关和链路的连接方式

总线

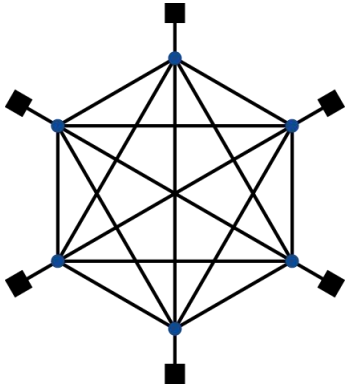
环



2D网格

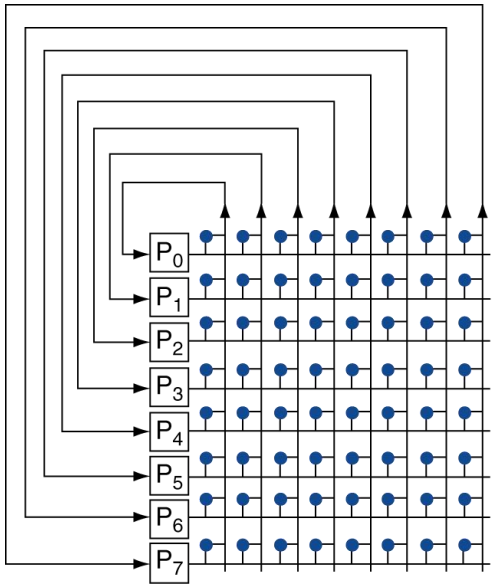


n-立方体(n = 3)

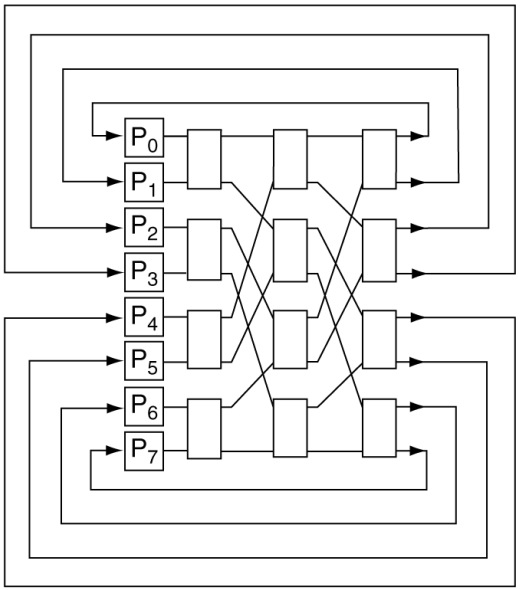


全连接

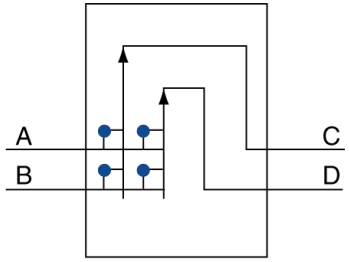
多级网络



a. Crossbar



b. Omega network



c. Omega network switch box

网络的特征

- 性能
  - 每条消息的延迟（无负载网络）
  - 吞吐率
    - 链路带宽
    - 总网络带宽
    - 切分带宽
  - 拥塞延迟（取决于通信量）
- 成本
- 功耗
- 硅片上的可连通性

并行基准测试程序

- Linpack：矩阵线性代数
- SPECrate：SPEC CPU程序的并行运行版本
  - 任务级并行
- SPLASH：Stanford Parallel Applications for Shared Memory
  - 由核心程序和应用程序构成，强比例缩放
- NAS (NASA Advanced Supercomputing)套件
  - 流体动力学计算核心
- PARSEC (Princeton Application Repository for Shared Memory Computers)套件
  - 采用Pthreads和OpenMP的多线程应用程序

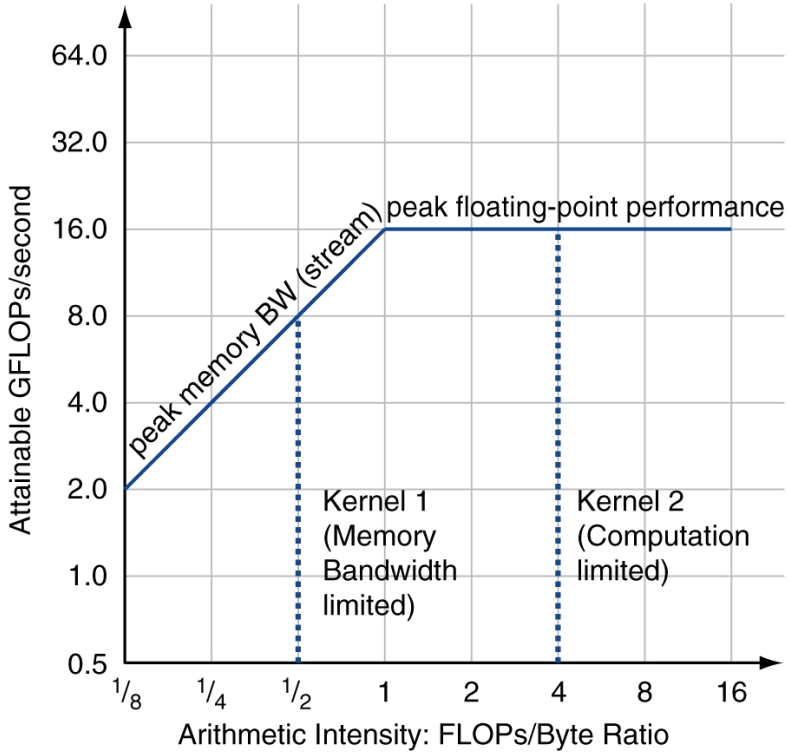
代码还是应用程序？

- 传统基准测试程序
  - 固定的代码和数据集
- 并行程序在不断演化
  - 算法、编程语言和工具该成为系统的一部分吗？
  - 让系统执行给定的应用程序，对比这些系统
  - 例如，Linpack、Berkeley Design Patterns
- 将促进并行化方法的革新

对性能建模

- 假定感兴趣的性能指标是可达到GFLOPs/秒
  - 用Berkeley Design Patterns中的计算核心度量
- 一个核心的算术密度
  - 每字节存储器访问包含的FLOPs
- 对于一个给定的计算机，测定
  - 峰值GFLOPs（来自数据手册）
  - 峰值存储器字节/秒（使用Stream基准测试程序）

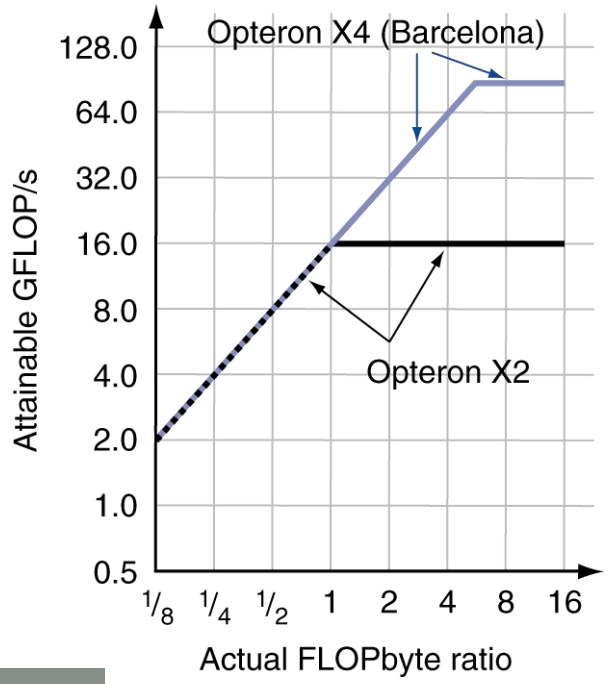
Roofline图



可达到的GFLOPs/秒
= Min (峰值存储器带宽 × 算术密度，峰值浮点性能)

比较不同系统

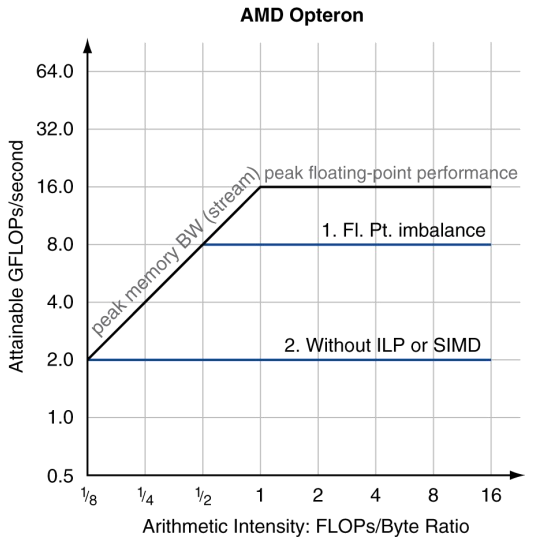
- 例：Opteron X2与Opteron X4
  - 2核与4核，2×浮点性能/核，2.2GHz与2.3GHz
  - 存储器系统相同

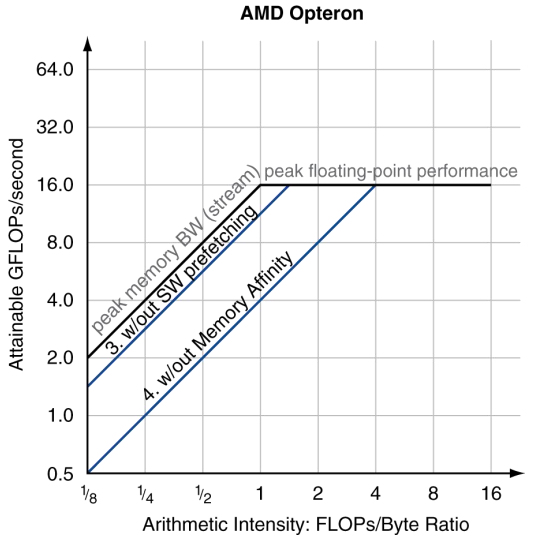


- 为了在X4上得到比X2更高的性能
  - 需要高算术密度
  - 或者工作集必须能存入X4的2MB L-3 cache

优化性能

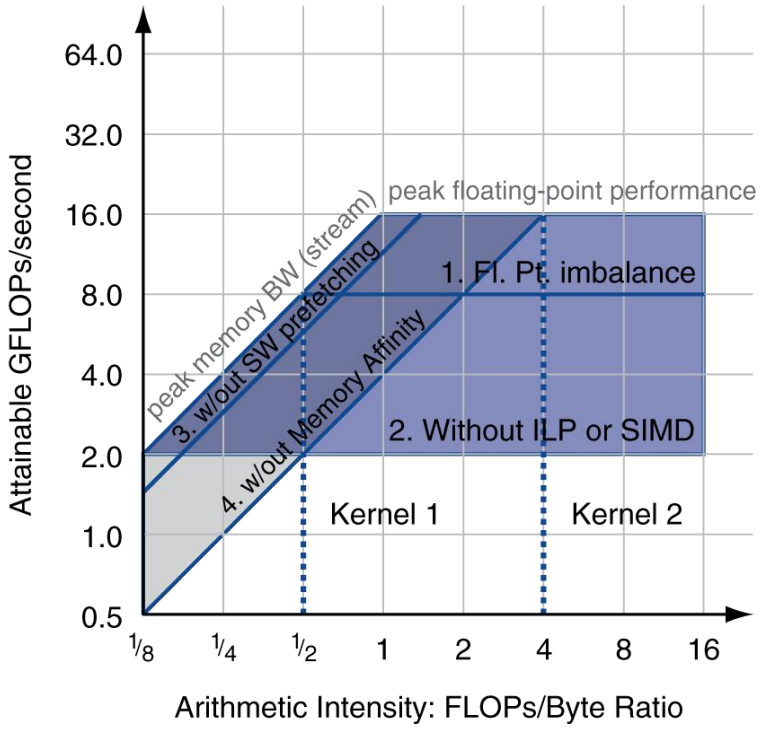
- 优化浮点性能
  - 平衡加法与乘法
  - 提高超标量ILP并使用SIMD指令
- 优化存储器用法
  - 软件预取
    - 避免取数阻塞
  - 存储器关联
    - 避免非本地数据访问





优化性能

- 优化方式的选择取决于代码的算术密度

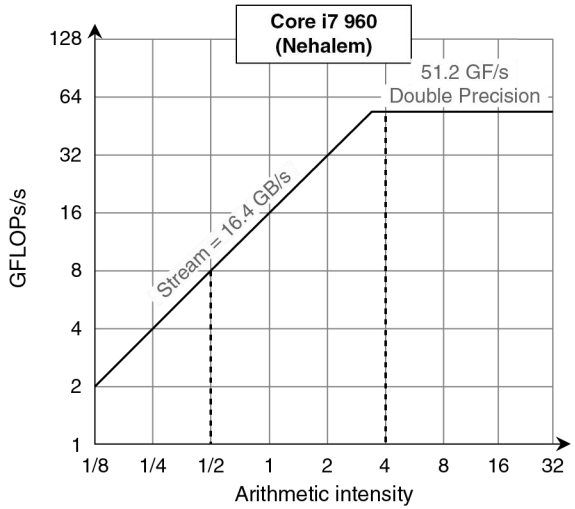


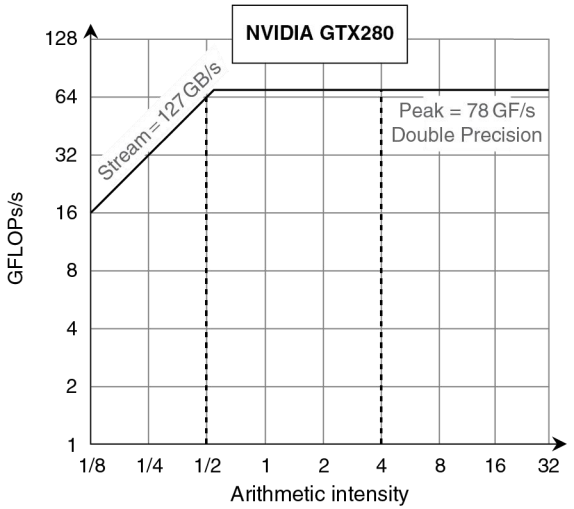
- 算术密度不总固定
  - 可随问题规模缩放
  - 缓存减少了存储器访问
    - 提高了算术密度

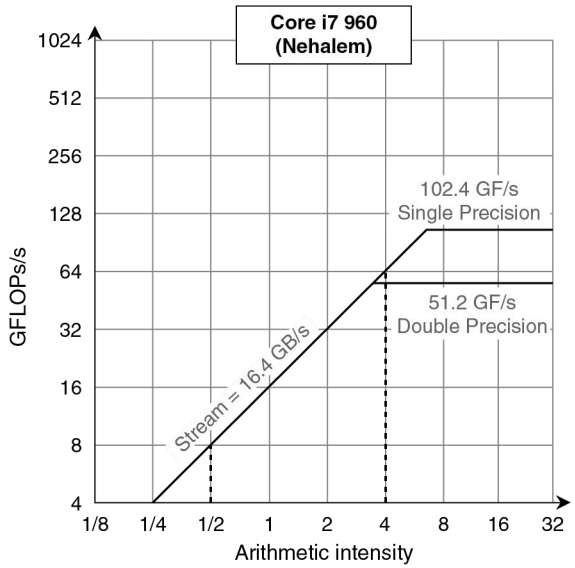
i7-960 vs. NVIDIA Tesla 280/480

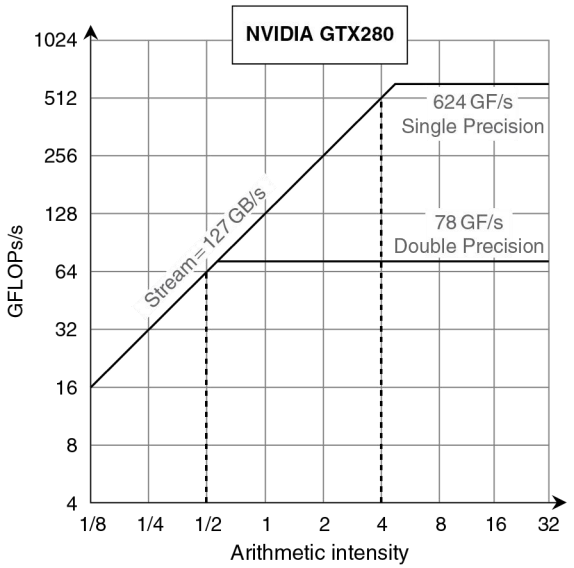
	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TCMS 65 nm	TCMS 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3100 M	2.0	4.4
Memory brandwith (GBytes/sec)	32	141	177	4.4	5.5
Single frecision SIMD width	4	8	32	2.0	8.0
Dobule precision SIMD with	2	1	16	0.5	8.0
Peak Single frecision scalar FLOPS (GFLOP/sec)	26	117	63	4.6	2.5
Peak Single frecision s SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 to 1344	3.0-9.1	6.6-13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused)	N.A	(622)	(1344)	(6.1)	(13.1)
(face SP dual issue fused)	N.A	(933)	N.A	(9.1)	–
Peal double frecision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

Roofline模型


Core i7 960
(Nehalem)
51.2 GF/s
Double Precision
Stream = 16.4 GB/s
GFLOPs/s
128
64
32
16
8
4
2
1
1/8
1/4
1/2
1
2
4
8
16
32
Arithmetic intensity


NVIDIA GTX280
Stream = 127 GB/s
Peak = 78 GF/s
Double Precision
GFLOPs/s
128
64
32
16
8
4
2
1
1/8
1/4
1/2
1
2
4
8
16
32
Arithmetic intensity


Core i7 960
(Nehalem)
102.4 GF/s
Single Precision
51.2 GF/s
Double Precision
Stream = 16.4 GB/s
GFLOPs/s
1024
512
256
128
64
32
16
8
4
1/8
1/4
1/2
1
2
4
8
16
32
Arithmetic intensity


NVIDIA GTX280
624 GF/s
Single Precision
78 GF/s
Double Precision
Stream = 127 GB/s
GFLOPs/s
1024
512
256
128
64
32
16
8
4
1/8
1/4
1/2
1
2
4
8
16
32
Arithmetic intensity

跑分

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

性能总结

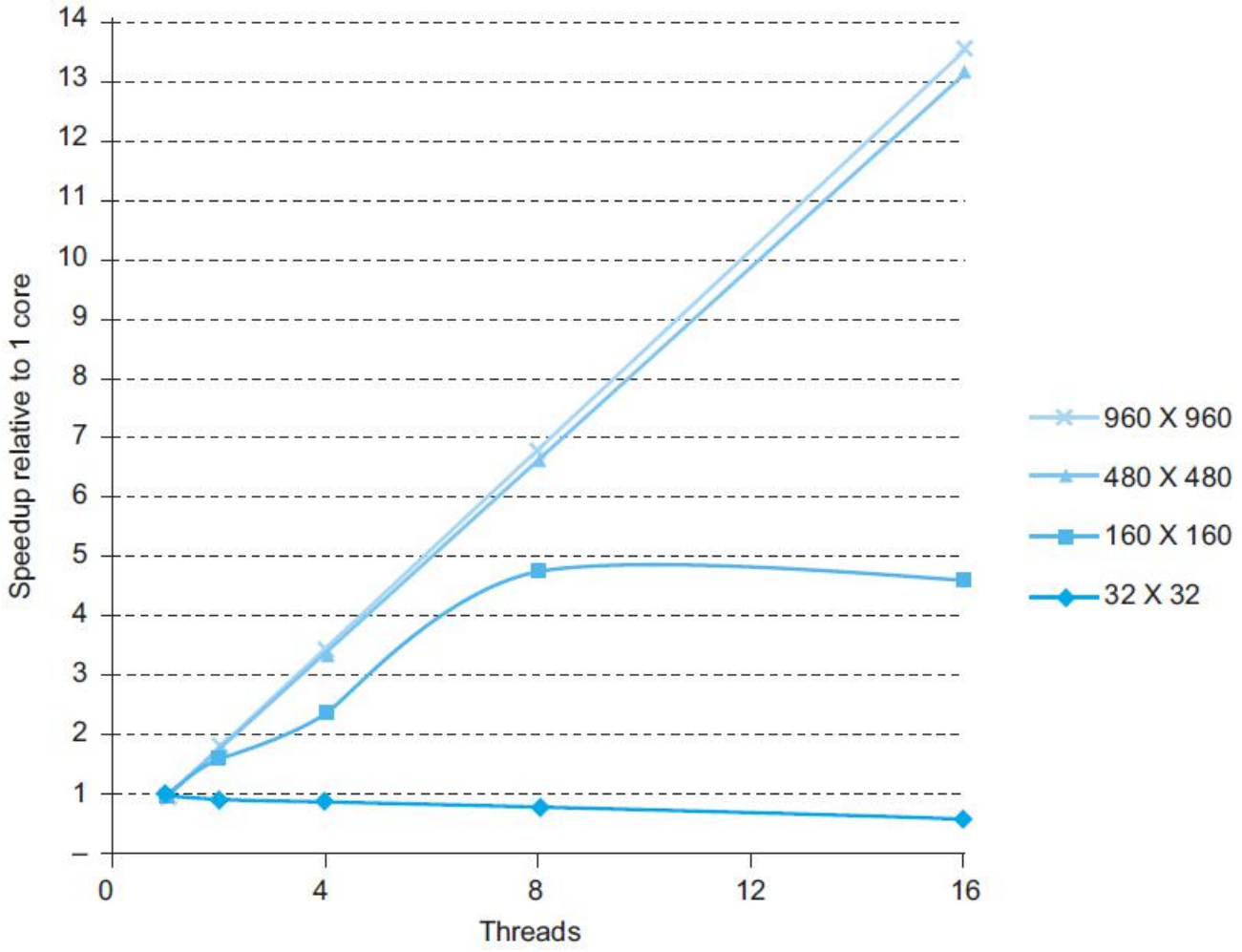
- GPU (480)有4.4倍存储器带宽
  - 有利于受存储器约束的核心程序
- GPU有13.1倍单精度吞吐率，2.5倍双精度吞吐率
  - 有利于受浮点计算能力约束的核心程序
- CPU的cache使某些核心程序变得不受存储器约束，而这些核心程序在GPU上仍受存储器约束
- GPU提供分散-聚集(scatter-gather)功能，对按一定步长访问数据的核心程序有帮助
- GPU缺乏同步和存储器一致性，限制了某些核心程序的性能

多线程DGEMM

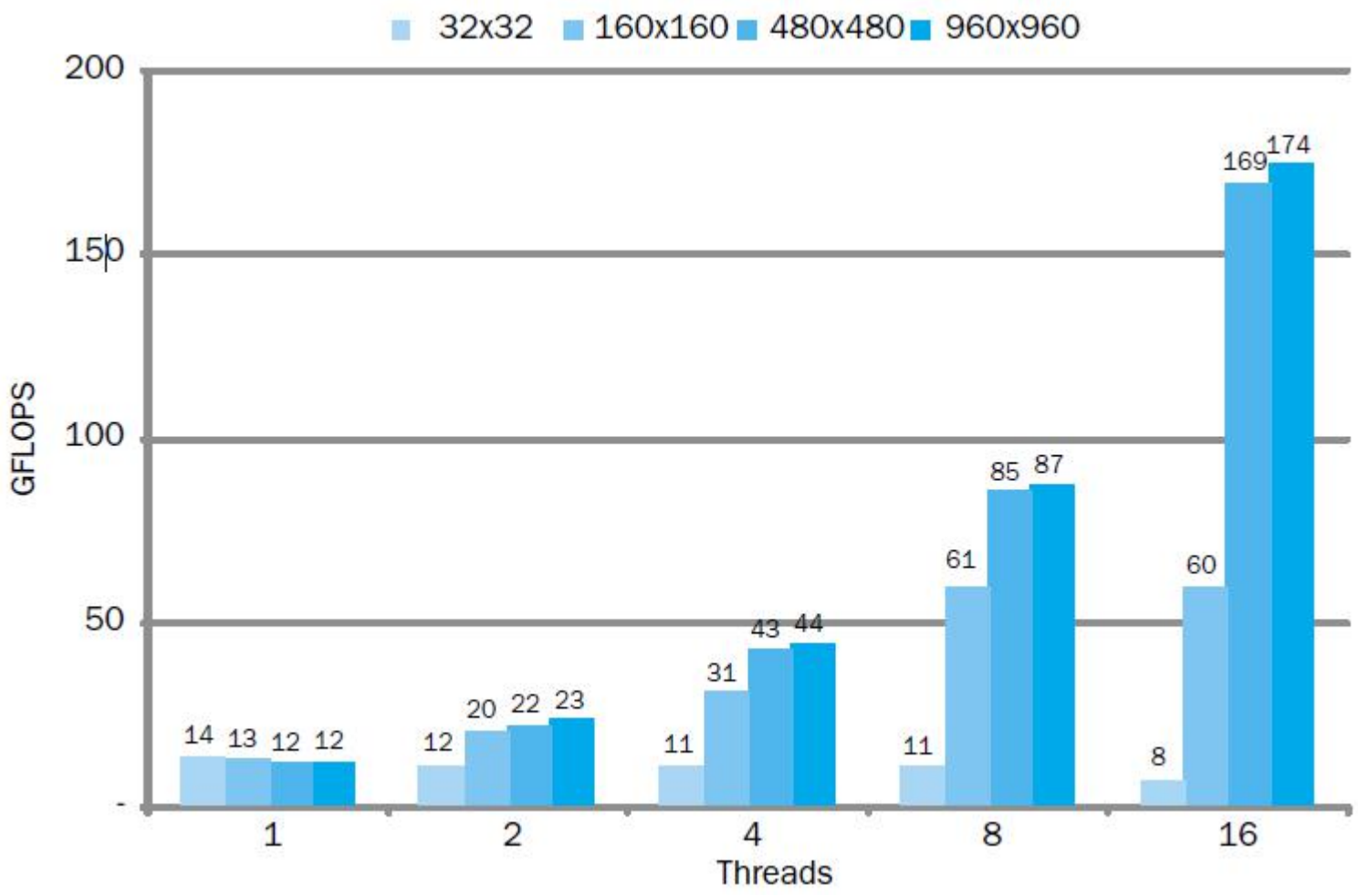
- 使用OpenMP：

```
void dgemm(int n, double* A, double* B, double* C)
{
#pragma omp parallel for
 for(int sj = 0; sj < n; sj += BLOCKSIZE)
  for(int si = 0; si < n; si += BLOCKSIZE)
   for(int sk = 0; sk < n; sk += BLOCKSIZE)
    do_block(n, si, sj, sk, A, B, C);
}
```

多线程DGEMM


Speedup relative to 1 core
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
4
8
12
16
Threads
960 X 960
480 X 480
160 X 160
32 X 32

多线程DGEMM


32x32
160x160
480x480
960x960
GFLOPS
200
150
100
50
-
14 13 12 12
12 20 22 23
11 31 43 44
11 61 85 87
8 60 169 174
1
2
4
8
16
Threads

谬误

- Amdahl定律不适用于并行计算机
  - 因为我们可以取得线性加速
  - 但仅限弱比例缩放下的应用程序
- 峰值性能和实际性能一致
  - 市场人员喜欢这种方法！
  - 但对比示例中的Xeon与其他处理器
  - 需要小心瓶颈

陷阱

- 在开发软件时不重视多处理器体系结构
  - 例：为共享的复合资源使用单个锁
    - 将访问串行化，即便可以并行访问
    - 用更细粒度的锁

本章小结

- 目标：通过使用多处理器获得更高性能
- 困难
  - 开发并行软件
  - 设计合适的体系结构
- SaaS（软件即服务）变得日益重要，并且集群与SaaS很匹配
- 性价比和能耗比推动着移动计算机和仓储级计算机的发展

本章小结（续）

- SIMD和向量操作适合多媒体应用，并且易于编程